PROBABILISTIC AND DIFFERENTIABLE PROGRAMMING

V7: Automatic Differentiation (AD)

Özgür L. Özçep
Universität zu Lübeck
Institut für Informationssysteme

Today‘s Agenda

$$\frac{\partial}{\partial x}\left(\texttt{</>} \right)$$

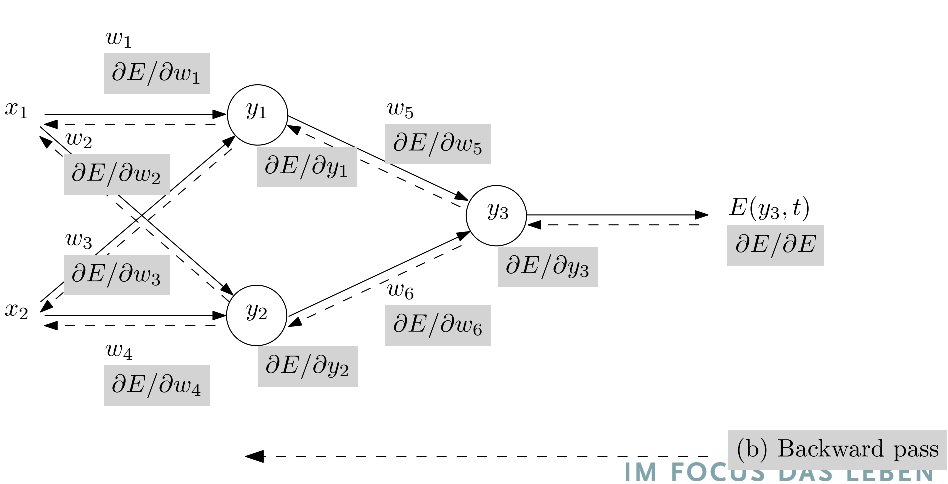
WHY YOU NEED AD

Reminder: Backprop = AD in reverse mode

Sample labeled data (**batch**) → **Forward** it through the network, get predictions → **Back-propagate** the errors → **Update** the network weights → (back to Sample labeled data)

Backpropagation idea

- Generate **error signal** that measures difference between predictions and target values
- Use error signal to change the weights and get more accurate predictions backwards
- Underlying mathematics: chain rule



Chain rule (1-dim)

$$\frac{dh}{dx} = \frac{df}{dg}\,\frac{dg}{dx}$$

(for $h(x) = f(g(x))$)

Reminder: Computational graph perspective

Function f

$$f(x,y,z) = (x+y) \cdot \mathrm{z}$$
$$= \mathrm{qz}$$
$$\text{for } \mathrm{q} = x + \mathrm{y}$$

Partial Derivatives

$$\frac{\partial f}{\partial z} = q \qquad \frac{\partial f}{\partial q} = z$$
$$\frac{\partial q}{\partial x} = 1 \qquad \frac{\partial q}{\partial y} = 1$$

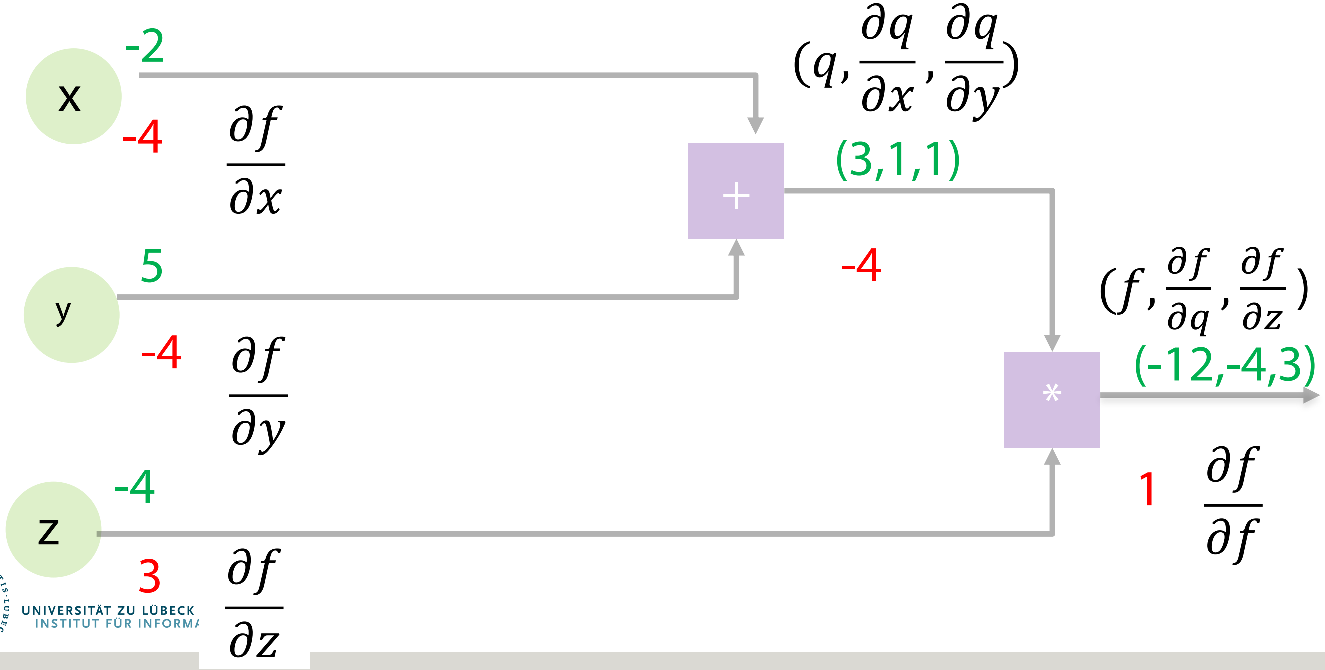
Chain rule applied

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z$$

Gradient

$$\nabla_{x,y,z} f = (z, z, q)$$

(In particular: $(\nabla_{x,y,z} f)(-2{,}5,-4) = (-4,-4,3)$)



Forward pass: function values and local gradients

Backward: chain rule

- To solve optimisation problems using gradient methods we need to compute the gradients (derivatives) of the objective with respect to the parameters.
  - In neural nets we're talking about the gradients of the loss function, L with respect to the parameters $\boldsymbol{\theta}$
  - AD is at the heart of „Differentiable Programming“ (the next big thing after deep learning)
    - AD is a topic on its own
    - But has been come into focus with Differentiable Programming and lead to many develeopments in the intersection of programming languages, numerical computing, and ML

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences

$$f'(a) \approx \frac{f(a+he_i)-f(a)}{h}$$

- Use Automatic Differentiation

- Problem: Static, expression swell. Can‘t differentiate algorithms
- Problem: Numerical errors (such as rounding and truncation errors)

Problem with symbolic computation

$\frac{d(f(x)\cdot g(x))}{dx} = \frac{d\,f(x)}{dx} g(x) + \frac{d\,g(x)}{dx} f(x)$ (Product rule)

- $h(x) := g(x) \cdot f(x)$
- $\frac{dh(x)}{dx}$ and h have two components in common
- This may also be the case for f.
- Symbollicaly calculating f won‘t profit from common parts of f and $\frac{df(x)}{dx}$

Problems with numerical calculation

Truncation error: Approximation error due to not sufficiently small h

- tends to 0 for $h \rightarrow 0$

Can be mitigated partly by using centered approximation

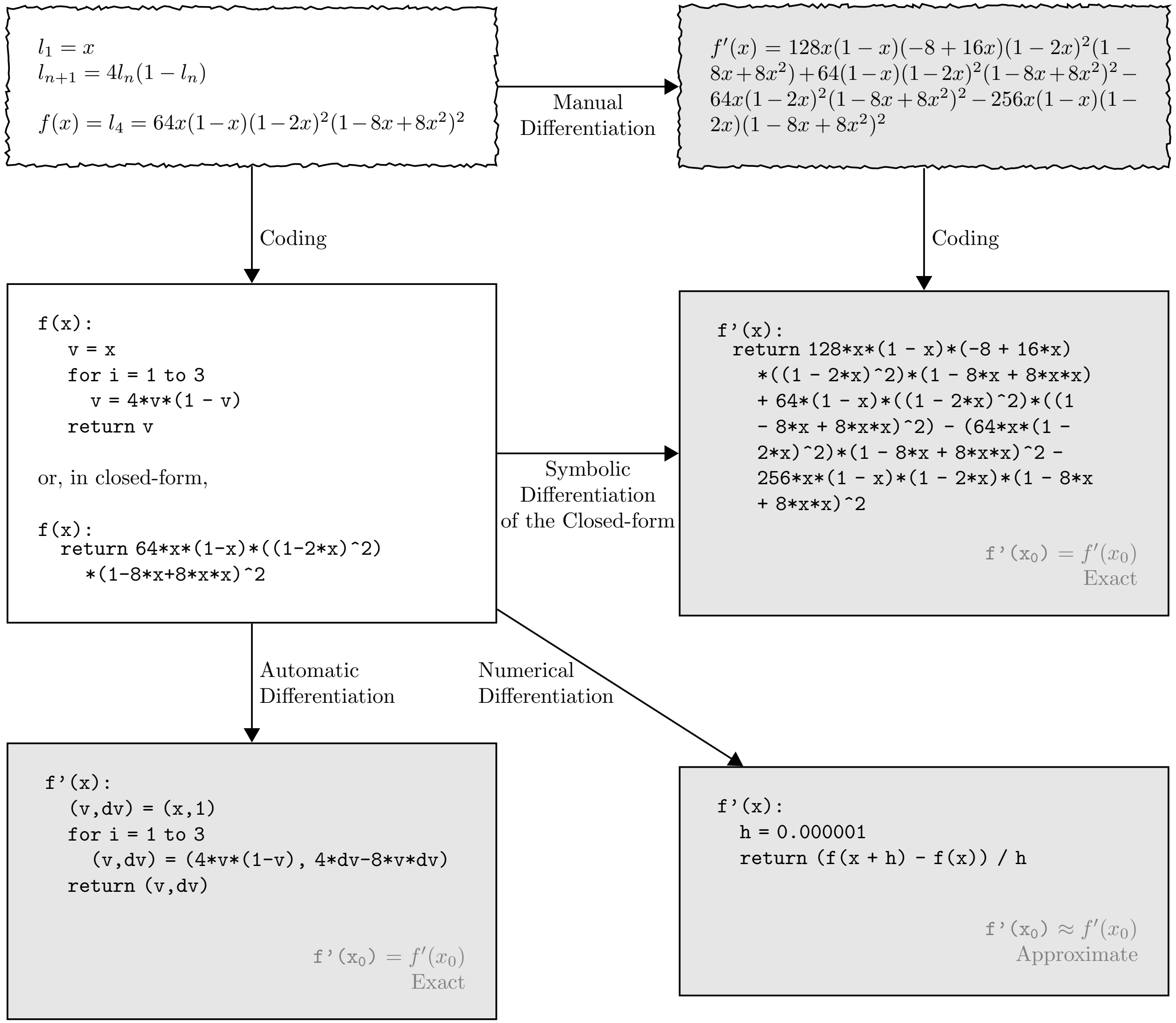
$$f'(a) \approx \frac{f(a+he_i)-f(a-he_i)}{h}$$

(error shift from $O(h)$ to $O(h^2)$)

Rounding error: due to limited precision in computation

- Increases for $h \rightarrow 0$

- Automatic Differentiation is a method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards.
  - Takes code that computes a function and uses that to compute the derivative of that function.
  - The goal isn’t to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.



$l_1 = x$

$l_{n+1} = 4l_n(1-l_n)$

$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$

Manual Differentiation

$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

Coding

Coding

```
f(x):
   v = x
   for i = 1 to 3
     v = 4*v*(1 - v)
   return v
```

or, in closed-form,

```
f(x):
  return 64*x*(1-x)*((1-2*x)^2)
    *(1-8*x+8*x*x)^2
```

Symbolic Differentiation of the Closed-form

```
f'(x):
 return 128*x*(1 - x)*(-8 + 16*x)
   *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
   + 64*(1 - x)*((1 - 2*x)^2)*((1
   - 8*x + 8*x*x)^2) - (64*x*(1 -
   2*x)^2)*(1 - 8*x + 8*x*x)^2 -
   256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
   + 8*x*x)^2
```

`f'(x₀)` $= f'(x_0)$
Exact

Automatic Differentiation

Numerical Differentiation

```
f'(x):
  (v,dv) = (x,1)
  for i = 1 to 3
    (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
  return (v,dv)
```

`f'(x₀)` $= f'(x_0)$
Exact

```
f'(x):
  h = 0.000001
  return (f(x + h) - f(x)) / h
```

`f'(x₀)` $\approx f'(x_0)$
Approximate

No one has time for manual computation

Gives you smell of the expression swell

Can directly reuse program with for-loop - no need for closed-form

Small h (as we have seen) does not help w.r.t. rounding errors

Ex: Baydin et al. 2017

AUTOMATIZATION

From Differentiation to Programming

- Example (Math)
- Example (code)

$$x = ?$$
$$y = ?$$
$$a = xy$$
$$b = \sin(x)$$
$$z = a + b$$

```
x=?
Y= ?
a = x * y
b = sin(x)
z = a + b
```

The chain rule for vectors

Given functions f, g with

- $\mathbb{R}^m \overset{g}{\rightarrow} \quad \mathbb{R}^n \quad \overset{f}{\rightarrow} \quad \mathbb{R}$
- $\boldsymbol{x} \quad \mapsto \quad \boldsymbol{y} = g(\boldsymbol{x}) \mapsto \quad z = f(\boldsymbol{y})$

the chain rule leads to the partial derivatives

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

(in short form: $\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z$

where $\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)$ is the $n \ x \ m$ Jacobian matrix of g)

Let us rename for the following

Given functions f, g with

- $\mathbb{R}^m \xrightarrow{g} \mathbb{R}^n \xrightarrow{f} \mathbb{R}$
- t $\mapsto \boldsymbol{u} = g(\boldsymbol{x}) \mapsto w = f(\boldsymbol{u})$

the chain rule leads to the partial derivatives

$$\frac{\partial w}{\partial t} = \sum_j \frac{\partial w}{\partial u_j} \frac{\partial u_j}{\partial t}$$

w is some output variable from a family of outputs $\{w_i\}$ and u_j are the inputs variables w depends on.

Applying the chain rule

Example expression

$$x = ?$$
$$y = ?$$
$$a = xy$$
$$b = \sin(x)$$
$$z = a + b$$

Derivatives w.r.t. some yet to be given variable t

$$\frac{\partial x}{\partial t} = ?$$
$$\frac{\partial y}{\partial t} = ?$$
$$\frac{\partial a}{\partial t} = x\frac{\partial y}{\partial t} + y\frac{\partial x}{\partial t}$$
$$\frac{\partial b}{\partial t} = \cos x\frac{\partial x}{\partial t}$$
$$\frac{\partial z}{\partial t} = \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}$$

- If we substitute $t = x$ we get an algorithm for computing $\frac{\partial z}{\partial x}$.
- Choosing $t = y$ similarly gives $\frac{\partial z}{\partial y}$.

Translating to code

Derivatives as programs

```
dx=?
dY= ?
da = y * dx + x * dy
db = cos(x)*dx
dz = da + db
```

Substituting $t = x$

```
dx= 1
dy= 0
da = y * dx + x * dy
db = cos(x)*dx
dz = da + db
```

(Using the notation

$\mathrm{dx} = \frac{\partial x}{\partial t}, \mathrm{dy} = \frac{\partial y}{\partial t}, \ldots$)

So, to compute $\frac{\partial z}{\partial x}$ just seed algorithm with

```
dx = 1, dy = 0
```

Translating to code

Derivatives as programs

```
dx=?
dY= ?
da = y * dx + x * dy
db = cos(x)*dx
dz = da + db
```

Substituting $t = y$

```
dx= 0
dy= 1
da = y * dx + x * dy
db = cos(x)*dx
dz = da + db
```

(Using the notation

$$\mathrm{dx} = \frac{\partial x}{\partial t}, \mathrm{dy} = \frac{\partial y}{\partial t}, \ldots$$)

So, to compute $\frac{\partial z}{\partial y}$ just seed algorithm with

```
dx = 0, dy = 1
```

Making Rules

- Idea of the examples can be generalized to arbitrary functions
- Need to describe rules for translation

program evaluating expression => program evaluating derivates

- These are just rules known from mathematics for calculating derivates, e.g.

```
- c = a + b           =>    dc = da + db
- c = a * b           =>    dc = b * da + a * db
- c = sin(a)          =>    dc = cos(a) * da
```

- Note: These rules are used on number-level (not for symbolic computation of derivatives)

Further Rules

```
c = a - b          =>     dc = da - db
c = a / b          =>     dc = da/b –a*db/b**2
c = a**b           =>     dc = b*a**(b-1)*da +
                               log(a)*a**b*db
c = cos(a)         =>     dc = -sin(a) * da
c = tan(a)         =>     dc = da/cos(a)**2
```

(`a**b` stands for a^b)

FORWARD MODE

Forward Mode AD

- To translate using the rules we simply replace each primitive operation in the original program by its differential analogue.
- The order of computation remains unchanged: if a statement K is evaluated before another statement L, then the differential analogue of K is evaluated before the analogue statement of L.
- This is Forward-mode Automatic Differentiation.
  - Nice feature: Interleaving (function evaluation and derivatives) is possible
  - Bad feature: Need to rerun program to compute derivative for each input (in particular for gradient)

Interleave computing expression and derivatives

```
x  =    ?
dx =    ?
y  =    ?
dy =    ?
a  = x * y
da = y * dx + x * dy
b  = sin(x)
db = cos(x)*dx
z  = a + b
dz = da + db
```

- Can keep track of value and gradient at the same time
- Can be mathematically founded using “dual numbers“
- Leads to direct simple implementation of AD

The Jacobian in Forward Mode AD

- $f\colon \mathbb{R}^n \to \mathbb{R}^m; \boldsymbol{x} \mapsto \boldsymbol{z}$
- Calculate derivatives w.r.t. ith variable x_i for all outputs z_i in one pass

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial z_1}{\partial x_1}(\boldsymbol{a}) & \boxed{\begin{matrix} \cdots \frac{\partial z_1}{\partial x_i}(\boldsymbol{a}) \\ \ddots \\ \cdots \frac{\partial z_m}{\partial x_i}(\boldsymbol{a}) \end{matrix}} & \begin{matrix} \frac{\partial z_1}{\partial x_n}(\boldsymbol{a}) \\ \vdots \\ \cdots\ \frac{\partial z_m}{\partial x_n}(\boldsymbol{a}) \end{matrix} \end{pmatrix}$$

$$\begin{matrix} \vdots \\ \frac{\partial z_m}{\partial x_1}(\boldsymbol{a}) \end{matrix}$$

- Efficient calculating product w.r.t. vector $\boldsymbol{r}$
  - $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \cdot \boldsymbol{r}$
  - Just seed with

    $dx_1 = r_1, \ldots, dx_n = r_n$

- Special case

  $f\colon \mathbb{R}^n \to \mathbb{R}; \boldsymbol{x} \mapsto z$

  Calculate directional derivate in direction $\boldsymbol{r}$.

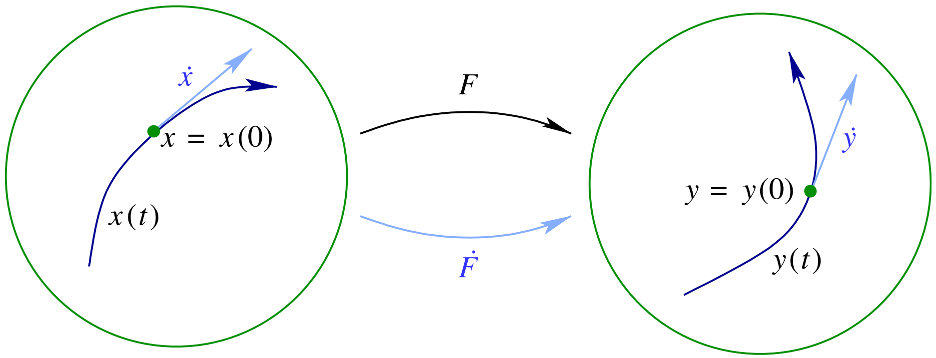
  - $\nabla f \cdot \boldsymbol{r}$

Another view on AD

- $f\colon \mathbb{R}^n \to \mathbb{R}^m$
  - $v_{i-n} = x_i, \mathrm{i} = 1, \dots, n$ input variables
  - $v_i,\quad \mathrm{i} = 1, \dots, l$ intermediate variables
  - $y_{m-i} = v_{l-i}, i = m - 1, \dots, 0$ output variables
  - $v_i = \phi_i\left(v_j\right)_{j \prec i}, \phi_i\colon \mathbb{R}^{n_j} \to \mathbb{R}$ (elemental functions)
    - where $\prec$ is precedence relation ($j \prec i$ iff v_i directly depends on v_j)
    - n_j number of elements preceding v_j
  - $u_i = \left(v_j\right)_{j \prec i}$

Forward mode AD = Tangents mapping

- Assume you have time-depending paths $x(t), y(t)$
- Forward mode AD is mapping function evaluation $(\mathrm{F}: \mathrm{x} \mapsto y)$ plus tangents mapping $\dot{F}: \dot{x} \mapsto \dot{y}$



$$[v_{i-n}, \dot{v}_{i-n}] = [x_i, \dot{x}_i] \qquad \text{for } i = 1, \ldots, n$$

$$[v_i, \dot{v}_i] = [\phi_i(u_i), \dot{\phi}_i(u_i, \dot{u}_i)] \qquad \text{for } i = 1, \ldots, l$$

$$[y_{m-i}, \dot{y}_{m-i}] = [v_{l-i}, \dot{v}_{l-i}] \qquad \text{for } i = 0, \ldots, m-1$$

Dual numbers (Clifford 1873)

- Want to mathematize parallel evaluation of f, f'
- Dual numbers have the form $(v + \dot{v}\epsilon)$ where
  - $v, \dot{v} \in \mathbb{R}$
  - ϵ is a nilpotent element ($\epsilon^2 = 0, \epsilon \neq 0$)
  - compare with complex numbers $x + yi$ where $i^2 = -1$, which can be considered as pairs in $\mathbb{R}^2$ (more general: quaterions)
- Gives intended behaviour mirroring symbolic derivation
  - $(v + \dot{v}\epsilon) + (u + \dot{u}\epsilon) = (v + u) + (\dot{v} + \dot{u})\epsilon$
  - $(v + \dot{v}\epsilon)(u + \dot{u}\epsilon) = (vu) + (v\dot{u} + \dot{v}u)\epsilon$

Dual numbers (Clifford 1873)

- Can define functions f on dual numbers by

  $$f(v + \dot{v}\epsilon) = f(v) + f'(v)\dot{v}\epsilon$$

  (results from Taylor series application)

- Then: Chain rule works as expected:

  $$\begin{aligned} f(g(v + \dot{v}\epsilon)) &= f(g(v) + g'(v)\dot{v}\epsilon) \\ &= f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon \end{aligned}$$

- Can extract derivative

  $$\frac{df}{dx}(v) = \epsilon - coeff(dual - version(f)(v + 1\epsilon))$$

REVERSE MODE

Reverse Mode AD

- Whilst Forward-mode AD is easy to implement, it comes with a very big disadvantage. . .
- For every variable we wish to compute the gradient with respect to, we have to run the *complete program* again.
- This is obviously going to be a problem if we're talking about the gradients of a function with very many parameters (e.g. a deep network).
- A solution is Reverse Mode Automatic Differentiation.

Reversing the Chain Rule

- Conceptually, chain rule doesn‘t care about role of enumerator and denominator – can turn it upside down
  - $\frac{\partial w}{\partial t}$ becomes
  - $\frac{\partial t}{\partial w}$ becomes by renaming (s for t and u for w)
  - $\frac{\partial s}{\partial u}$ is by applying chain
  - $\frac{\partial s}{\partial u} = \sum_j \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i}$
    - u is some input variable
    - w_is are output variables depending on u
    - s is the yet-to-be-given variable
- Now can compute in 1-pass in parallel: $\frac{\partial s}{\partial x}, \frac{\partial s}{\partial y}, \ldots$

Example

$$\frac{\partial s}{\partial u} = \sum_j \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i}$$

$$x = ?$$
$$y = ?$$
$$a = xy$$
$$b = \sin(x)$$
$$z = a + b$$

$$\frac{\partial s}{\partial z} = ?$$

$$\frac{\partial s}{\partial b} = \frac{\partial z}{\partial b} \frac{\partial s}{\partial z} = \frac{\partial s}{\partial z}$$

$$\frac{\partial s}{\partial a} = \frac{\partial z}{\partial a} \frac{\partial s}{\partial z} = \frac{\partial s}{\partial z}$$

$$\frac{\partial s}{\partial y} = \frac{\partial a}{\partial y} \frac{\partial s}{\partial a} = x \frac{\partial s}{\partial a}$$

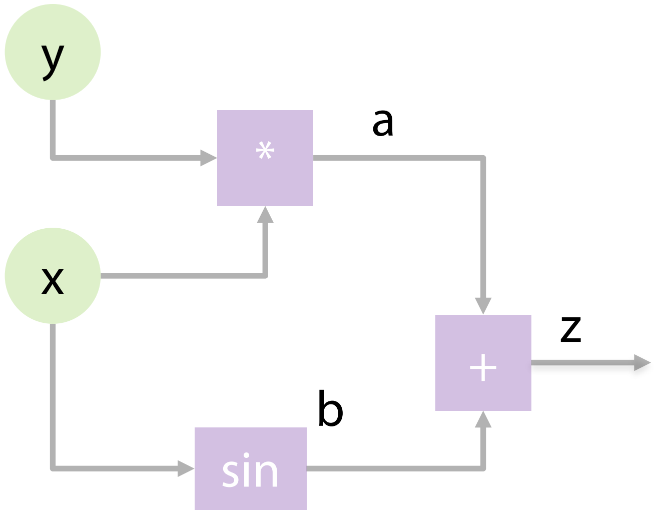
$$\frac{\partial s}{\partial y} = \frac{\partial a}{\partial x} \frac{\partial s}{\partial a} + \frac{\partial b}{\partial x} \frac{\partial s}{\partial b}$$

$$= y \frac{\partial s}{\partial a} + \cos x \frac{\partial s}{\partial b}$$

$$= (y + \cos x) \frac{\partial s}{\partial z}$$

Visualising dependencies

- Differentiating in reverse can be quite mind-bending: instead of asking what input variables an output depends on, we have to ask what output variables a given input variable can affect.
- We can see this visually by drawing a dependency graph of the expression (e.g. x effects a and b):



Translating to Code

- As before we replace the derivatives $(\partial s/\partial z, \partial s/\partial b, \ldots)$ with variables `(gz, gb, ...)` which we call adjoint variables:

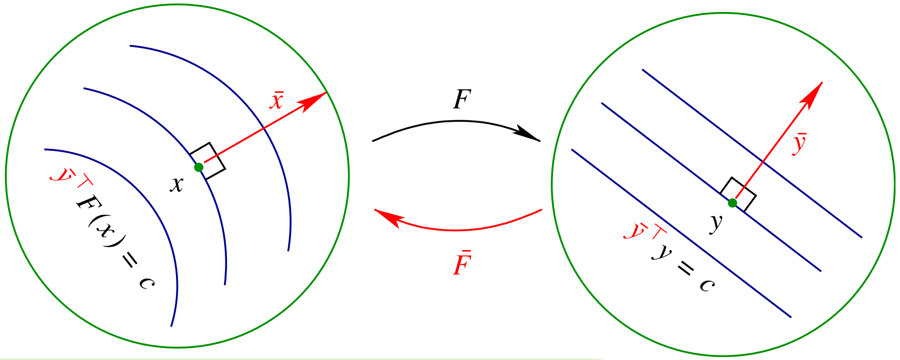
```
– gz = ?
– gb = gz
– ga = gz
– gy = x * ga
– gx = y * ga + cos(x) * gb
```

We need only 1 pass for Calculating all derivatives

- Substituting $s = z$ in equations gives both gradients $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ in last two lines
- Equivalently set gz = 1

Reverse mode AD = Co-tangents mapping

- Reverse mode AD: function evaluation, $F\colon x \mapsto y$, plus co-tangents mapping by adjoint $\bar{F}\colon \bar{y} \mapsto \bar{x}$



$$\bar{F}(x, \bar{y}) \coloneqq \bar{y}^\top F'(x) = \bar{x}$$

```
$v_i = 0$                                   for $i = 1, \dots, l$
$[v_{i-n}, \bar{v}_{i-1}] = [x_i, \bar{x}_i]$      for $i = 1, \dots, n$
$Push(v\_i)$
$v_i = \phi_i(u_i)$                         for $i = 1, \dots, l$
$y_{m-1} = v_{l-1}$                         for $i = 0, \dots, m-1$
$\bar{v}_{l-i} = \bar{y}_{m-1}$                   for $i = 0, \dots, m-1$
$v_i \leftarrow pop(\ )$
$\bar{u}_i \mathrel{+}= \bar{v}_i * \nabla\phi_i(u_i)$          for $l, \dots, 1$
$v_i = 0$
$\bar{x}_i = \bar{v}_{i-n}$                       for i $= 1, \dots, n$
```

(simple algorithm without sophisticated memory management: just using stack)

But wait... Limitations of Reverse Mode AD

- We have a problem dual to that of forward AD: Now have to run the program for each outvariable one is interested in differentiating

- Example
  - $z = 2x + \sin x$
  - $v = 4x + \cos x$
  - Cannot interleave the calculations as they appear to be in reverse mode.=> Recent research on automatization

Calculating $\frac{\partial z}{\partial x}$ and $\frac{\partial v}{\partial x}$ each requires running the programm

- So: Reverse AD has advantage only if number of output variables much smaller than number of input variables

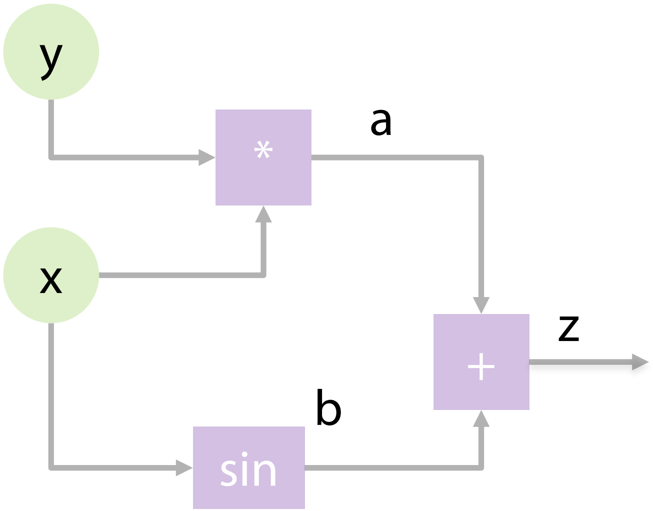
Implementing Reverse Mode AD

There are two ways to implement Reverse AD:

1. We can parse the original program and generate the *adjoint* program that calculates the derivatives.
   - Potentially hard to do.
   - Static, so can only be used to differentiate algorithms that have parameters predefined.
   - But, efficient (lots of opportunities for optimisation)
2. We can make a *dynamic* implementation by constructing a graph that represents the original expression as the program runs.

Constructing an expression graph (in Python)

- Goal: get a graph as

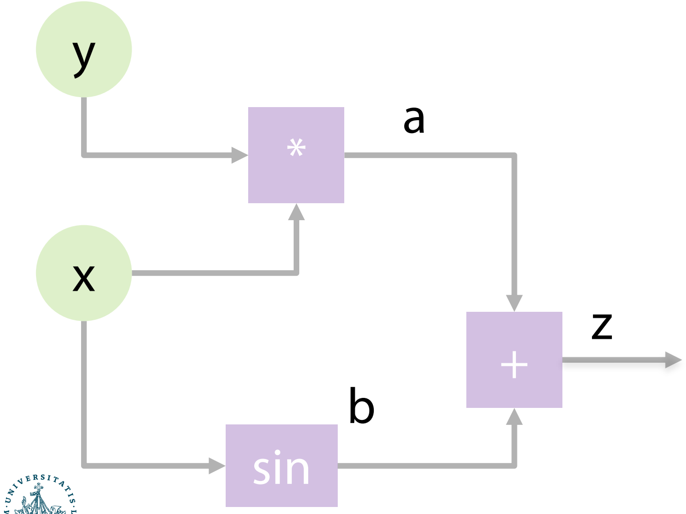


- Root of the graph are independent variables x, y
- Can have children (initially empty): nodes that depending on parent

```
class Var:
def __init__(self, value):
self.value = value
self.children = [] ...
...
x  = Var (0.5)
y  = Var (4.2)
```

Building expressions

- Expression creation
- Self-registration of each expression u as a child of each of its dependencies w_i
- Also register weight $\frac{\partial w_i}{\partial u}$ (used for gradient calculation)

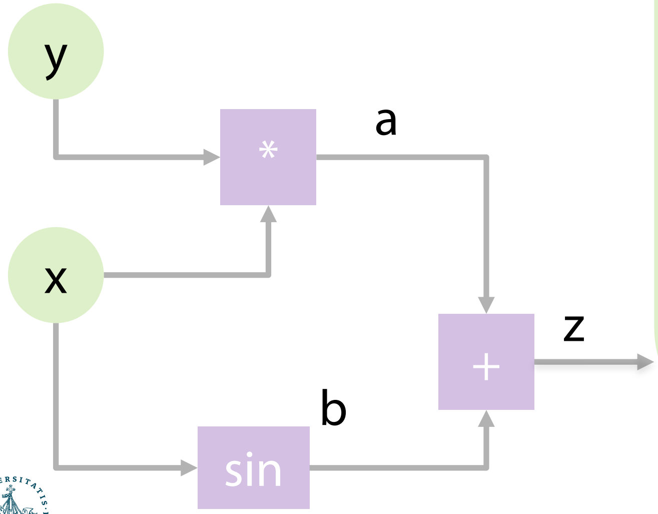


```
class Var:
...
def __mul__(self, other):
  z = Var(self.value * other.value)

  # weight = dz/dself = other.value
  self.children.append((other.value, z))

  # weight = dz/dother = self.value
  other.children.append((self.value, z))
  return z
...
...

# "a" is a new Var that is a child of
both x and y
#  a=x*y
```

Computing gradients

- Propagate Derivatives
- Cache derivatives in `grad_value`

```
class Var:
def __init__(self):
...
self.grad_value = None

def grad(self):
  if self.grad_value is None:
  # using chain rule
  self.grad_value =
    sum(weight * var.grad()
         for weight, var in self.children)
return self.grad_value ...
...
a.grad_value = 1.0
print("da/dx␣=␣{}".format(x.grad()))
```



Optimising reverse Mode AD

- The outline implemntation not very space efficient
  - Instead of children direclty store in indices (Wengert list, tape)
- Space efficiency for reverse AD is challenging hence research topic
  - Count-Trailing-Zeros CTZ): trade-off computation for memory of caches (Griewank 92).
  - But, in reality memory is relatively cheap (if managed well)

CTZ example

- Idea: Hierarchical cache storing only $O(\log(N))$ of all N values in expression in forward sweep and maintained during reverse sweep
  - Cache$_0$ store first value
  - Cache$_1$ stores value at ½ down chain
  - Cache$_2$ stores value at ¾ down the chain …
  - Cache$_{n-1}$ stores value at n/n+1 down the chain
- Assume linear expression of N= 16 Operators
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f (value indices)
  - X----------------X-------X---X X (stored value indication)

CTZ example (continued)

- Reverse sweep (with head postion _)
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f _ (can swee over e,f)
  - X-----------------X-------X---X X

  - 0 1 2 3 4 5 6 7 8 9 a b c d e f (d not cached, recalculate)
  - X-----------------X-------X---X X

  - 0 1 2 3 4 5 6 7 8 9 a b c d e f (d not cached, recalculate)
  - X-----------------X-------X---X X from cached c)

    +-X

CTZ example (continued)

- Reverse sweep (with head postion _)
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f
  - X-----------------X-------X---X X

    +-X

    (sweep over c, missing b,
    sweep to 8 **and** cache a)
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f
  - X-----------------X-------X---X X
  - +--X-X+-X

    (recompute 9 from 8,
    then 7to be recomputed
    move to 0, store along 6,4)
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f
  - X-----------------X-------X---X X
  - +-------X--X-X-+---X-X+-X
  - +-X

    (and so on ...)

CTZ example (continued)

- In the end
  - 0 1 2 3 4 5 6 8 9 a b c d e f
  - X-----------------X-------X---X X
  - +-------X--X-X-+---X-X+-X
  - +---X-X   +-X
  - +-X

Uhhh, a lecture with a hopefully useful

APPENDIX

Color Convention in this Course

- Formulae, when occurring inline
- Newly introduced terminology and definitions 🟦
- Important results (observations, theorems) as well as emphasizing some aspects 🟥
- Examples are given with standard orange with possibly light orange frame 🟧
- Comments and notes in nearly opaque post-it 🟨
- Algorithms and program code 🟩
- Reminders (in the grey fog of your memory) ⬜

Today‘s lecture is based on the following

- Jonathon Hare: Lecture 5 of course „COMP6248 Differentiable Programming (and some Deep Learning)“ http://comp6248.ecs.soton.ac.uk/
- Blog post by Rufflewind: Reverse-mode automatic differentiation: a tutorial https://rufflewind.com/2016- 12- 30/reverse- mode- automatic-differentiation
- A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: A survey. J. Mach. Learn. Res., 18(1):5595–5637, Jan. 2017.
- A. H. Gebremedhin and A. Walther. An introduction to algorithmic differentiation. WIREs Data Mining and Knowledge Discovery, 10(1):e1334, 2020.

References

- W. K. Clifford. Preliminary sketch of bi-quaternions. Proceedings of the London Mathematical Society, pages 381—395, 1873.
- A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optimization Methods and Software, 1(1):35–54, 1992.